APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/024,961	12/19/2001	David Pociu	13729-003001	3454

26161	7590	04/07/2005

FISH & RICHARDSON PC
225 FRANKLIN ST
BOSTON, MA 02110

EXAMINER
MOFIZ, APU M

ART UNIT	PAPER NUMBER
2165	

DATE MAILED: 04/07/2005

Please find below and/or attached an Office communication concerning this application or proceeding.

PTO-90C (Rev. 10/03)

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE <u>3</u> MONTH(S) FROM
THE MAILING DATE OF THIS COMMUNICATION.
- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed
  after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133).
  Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any
  earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on <u>22 November 2004</u>.

2a)☒ This action is **FINAL**.   2b)☐ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is
closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1,4-11,13-17,20,21,23,24,26 and 27* is/are pending in the application.

    4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1,4-11,13-17,20,21,23,24,26 and 27* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *19 December 2001* is/are: a)☒ accepted or b)☐ objected to by the Examiner.

    Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

    Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

    a)☐ All b)☐ Some * c)☐ None of:

      1.☐ Certified copies of the priority documents have been received.

      2.☐ Certified copies of the priority documents have been received in Application No. _____.

      3.☐ Copies of the certified copies of the priority documents have been received in this National Stage
application from the International Bureau (PCT Rule 17.2(a)).

    * See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1)☒ Notice of References Cited (PTO-892)

2)☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)

3)☐ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08)
    Paper No(s)/Mail Date _____.

4)☐ Interview Summary (PTO-413)
    Paper No(s)/Mail Date. _____ .

5)☐ Notice of Informal Patent Application (PTO-152)

6)☐ Other: _____.

DETAILED ACTION

Examiner's Response to Applicant's Remarks

1.   Applicant's arguments submitted on 11/22/2004 with respect to claims 1,4-11,13-17,20-21,23-24,26-27 have been reconsidered but are not deemed persuasive for the reasons set forth below.

Examiner's Responses to Applicant's Remarks are listed below:

2.   Applicant argues (under REMARKS section) that, BEA WebLogic does not teach "A method comprising, in a network, encapsulating data requests generated by an application in a first system, encapsulating comprising generating an Extensible Markup Language (XML) structure for each data request and converting the XML structure to an XML request, the XML structure comprising a variable stream of data stored in memory of the first system, the stream including an XML element for each request; transferring the encapsulated data requests to a second system; executing the encapsulated data requests in the second system; and processing in the first system responses generated by the encapsulated data requests in the second system."

Examiner respectfully disagrees. BEA WebLogic teaches **a method comprising, in a network** (i.e., *"Extensible Markup Language (XML) is a markup language for documents containing structured information. It is a simplified version of Standard Generalized Markup Language (SGML) and has become an industry standard for delivering content on the Internet."*) (page 1; page 2), **encapsulating data**

requests (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents*

to the appropriate output stream. In the code segment below StockClient generates XML based on user input that

defines a stock trade transaction. " ... // send xml to servlet

    pw.println ("<?xml version='1.0' ?>") ;

    // set document type declaration

    pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL

      +"stocktrade.dtd'>");

. *// set stock trade instructions*

    pw.print("<stocktrade ");

    pw.print("action = "+ (buy ? "'buy' : : "'sell' "));

    pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));

    pw.print("numshares = '"+ line +"'");

    pw.println(" />");"

    ...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document

has been generated, you simply write the document to the appropriate output stream. " The preceding text excerpts

clearly indicate that a user most likely through a web browser application/ console application sends requests for

some stock transaction data to StockClient application. StockClient collects all of the necessary request information

e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to

StockServlet application through an output stream. The XML document includes elements i.e., data

enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a

structured XML request document in the first system to be sent to a second system through a stream, encapsulation.)

(Page 4; page 21) **generated by an application in a first system** (i.e. the system with the

StockClient application) (page 4), **encapsulating** (page 4) **comprising generating an**

Extensible Markup Language (XML) structure (page 4) **for each data request** (page 4)

and converting the XML structure (page 4) **to an XML request** (page 4), **the XML**

structure (page 4) **comprising a variable stream** (page 4) **of data stored in memory** (i.e.,

"The DOM API provides methods for building and modifying entire XML documents in memory) (page 4) **of the**

first system (page 4), **the stream** (page 4) **including an XML element** (page 4) **for each**

request (page 4); **transferring** (i.e., *"StockClient connects to the StockServlet and sends an XML document*

via the POST method. The following code segment from StockClient obtains the URL to the servlet from the

argument list, opens a URL connection, obtains the output stream from the connection, and prints the first couple of

lines of the XML document to the output stream. " The preceding text excerpts clearly indicate that StockClient

transfers the request document to the StockServlet.) (page 6) **the encapsulated** (page 4) **data requests**

(page 4) **to a second system** (page 4); **executing** (i.e., *"In this example, StockServlet acts as a mediator*

between StockClient and TraderBean. StockClient accepts user input to define a stock trade transaction. This data is

used to generate XML in the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST.

StockServlet creates an instance to RequestHandler to parse the XML using a validating SAX parser and operates

on TraderBean to execute the trade. StockServlet then receives the result of the trade from TraderBean, generates

XML in the format of traderresult.dtd, and sends the XML back to StockClient via a HTTPServletResponse.

StockClient parses the XML using a validating SAX parser and displays the results to the console. " The preceding

text excerpts clearly indicate that the second system i.e., the system/ WebLogic Server with StockServlet,

TraderBean etc. receives the transaction request data from the StockClient. The second system parses the data and

then the EJB TraderBean executes the data i.e., processes the transaction. The result is then sent back to the first

system, i.e., the system with StockClient. StockClient parses the result from the transaction result and shows to the

user. The stock transaction is a persistent transaction. Therefore, the TraderBean after processing the transaction

persists/ commits the data in a persistent storage e.g., a database.) (page 21; page 22; page 26) **the**

encapsulated data requests (page 4) **in the second system** (page 4); **and processing**

(page 21; page 22; page 26) **in the first system responses** (page 21; page 22; page 26) **generated**

by the encapsulated data requests (page 21; page 22; page 26) **in the second system** (page

21; page 22; page 26).

3.   Applicant argues (under REMARKS section) that, BEA WebLogic does not teach "A distributed application method comprising: converting application requests in a first system, converting comprising generating a data structure for storing data and parameters related to an application that produced the application requests, translating the application requests into a standardized delimited data structure in conjunction with the data structure into a stream of text based data utilizing a Extensible Markup Language (XML) format; transmitting the converted application requests to a second system over a network; parsing the converted application requests in the second system into request statements; and executing the request statements in the second system."

Examiner respectfully disagrees. BEA WebLogic teaches **a distributed application** (i.e., *"Extensible Markup Language (XML) is a markup language for documents containing structured information. It is a simplified version of Standard Generalized Markup Language (SGML) and has become an industry standard for delivering content on the Internet."*) (page 1; page 2) **method comprising: converting application requests** (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream. In the code segment below StockClient generates XML based on user input that defines a stock trade transaction."* ... // send xml to servlet

pw.println ("<?xml version='1.0' ?>") ;

// set document type declaration

pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL

   + "stocktrade.dtd'> ");

// set stock trade instructions

pw.print("<stocktrade ");

pw.print("action = "+ (buy ? "'buy' : : "'sell' "));

pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));

pw.print("numshares = '"+ line +"'");

pw.println(" />");"

...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document has been generated, you simply write the document to the appropriate output stream. " The preceding text excerpts clearly indicate that a user most likely through a web browser application/ console application sends requests for some stock transaction data to StockClient application. StockClient collects all of the necessary request information e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to StockServlet application through an output stream. The XML document includes elements i.e., data enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a structured XML request document in the first system to be sent to a second system through a stream, conversion of application requests.) (Page 4; page 21) **in a first system** (Page 4; page 21), **converting comprising generating a data structure** (Page 4; page 21) **for storing data and parameters related to an application that produced the application requests** (i.e., the XML structure created by StockClient stores all of the transaction request data sent by the user through a console application/ a web browser) (Page 4; page 21), **translating the application requests** (Page 4; page 21) **into a standardized delimited data structure** (Page 4; page 21) **in conjunction with the data structure** (Page 4; page 21) **into a stream** (i.e., the output stream) (Page 4; page 21) **of text based data utilizing a Extensible Markup Language (XML) format** (Page 4; page 21); **transmitting** (i.e.,

"StockClient connects to the StockServlet and sends an XML document via the POST method. The following code segment from StockClient obtains the URL to the servlet from the argument list, opens a URL connection, obtains the output stream from the connection, and prints the first couple of lines of the XML document to the output stream. " The preceding text excerpts clearly indicate that StockClient transmits/ transfers the request document to

the StockServlet.) (page 6) **the converted application requests** (Page 4; page 21) **to a second**

system (Page 4; page 21) **over a network** (page 1; page 2); **parsing** (i.e., *"In this example, StockServlet*

acts as a mediator between StockClient and TraderBean. StockClient accepts user input to define a stock trade

transaction. This data is used to generate XML in the format defined by stocktrade.dtd and the XML is sent to

StockServlet via a POST. StockServlet creates an instance to RequestHandler to parse the XML using a validating

SAX parser and operates on TraderBean to execute the trade. StockServlet then receives the result of the trade from

TraderBean, generates XML in the format of traderresult.dtd, and sends the XML back to StockClient via a

HTTPServletResponse. StockClient parses the XML using a validating SAX parser and displays the results to the

console. " The preceding text excerpts clearly indicate that the second system i.e., the system/ WebLogic Server with

StockServlet, TraderBean etc. receives the transaction request data from the StockClient. The second system **parses**

the data and then the EJB TraderBean **executes** the data i.e., processes the transaction. The result is then sent back to

the first system, i.e., the system with StockClient. StockClient parses the result from the transaction result and shows

to the user. The stock transaction is a persistent transaction. Therefore, the TraderBean after processing the

transaction persists/ commits the data in a persistent storage e.g., a database.) (page 21; page 22; page 26) **the**

converted application requests (Page 4; page 21) **in the second system** (Page 4; page 21)

into request statements (page 21; page 22; page 26); **and executing** (page 21; page 22; page 26)

the request statements (page 21; page 22; page 26) **in the second system** (page 21; page 22;

page 26).

4.   Applicant argues (under REMARKS section) that, BEA WebLogic does not teach

"An application server method comprising: generating a first data structure for storing

data and parameters related to an application residing in the server, the first data

structure comprising database tables, procedure results from logic calls and status/error

messages; translating application requests from the application into a delimited second

data structure, stored in a memory, the second data structure comprising an element for each of the application requests; generating a stream of text based, data in an Extensible Markup Language (XML) format from the second data structure."

Examiner respectfully disagrees. BEA WebLogic teaches **an application server** (i.e., the WebLogic server with TraderBean) (page 21; page 22; page 26) **method comprising: generating** (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream." ... "In this example, StockServlet acts as a mediator between StockClient and TraderBean. StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to* **execute** *the trade. StockServlet then receives the result of the trade from TraderBean,* **generates** *XML in the format of traderresult.dtd, and sends the XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating SAX parser and displays the results to the console." ... "Session Bean *Pure business "logic" bean *Contain no data *Similar to a* **stored procedure***, but able to take advantage of the object services. Entity Bean *Represent a row in the table: "data" bean *A* **business object** *with a unique business identity *Identify determined by the primary key. Combine entity and session Beans as appropriate. An example using an order entry system: *Customer entity bean: the customer record *Order entity bean:the service order *Order process session bean: knows the process for completing an order: 1. Check customer credit limit 2. Check availability of stock 3. Place order, record (commit) a transaction." ... "Though ejbStore() is called before the EJBean is actually written to the* **database***, in the event an* **exception** *is raised while writing to the database, the framework will assume that the EJBean has been modified, regardless of the setting of the flag."* The preceding text excerpts clearly indicate that the second system i.e., the system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the data i.e., processes the transaction. The result is then sent back to the first system, i.e., the

system with StockClient. StockClient parses the result from the transaction result and shows to the user. Therefore

whether it is a stock transaction or it is just getting a stock quote from the server, the TraderBean (whether it is a

session or entity bean) has to interact with a persistent storage. Whether the TraderBean is a session or entity bean it

performs some business logic for the transaction. The transaction result may be just getting the stock quote from the

persistent storage i.e., a database table. The transaction result may comprise **failure/error/exception** to commit the

transaction e.g., the requested stock is not available. The result may comprise an exception raised during the usage

of the business logic of the EJBean. The EJBean i.e., TraderBean sends the **application result i.e., first data**

structure to the StockServlet. The StockServlet converts the result into **a delimited XML structure i.e., second**

data structure to the StockClient, which stores application results for the StockClient requested data elements.)

(page 4; page 5; page 21; page 22; page 25; page 26; page 30) **a first data structure** (page 4; page 5; page

21; page 22; page 25; page 26; page 30) **for storing data and parameters** (page 4; page 5; page 21;

page 22; page 25; page 26; page 30) **related to an application residing in the server** (page 4; page

5; page 21; page 22; page 25; page 26; page 30), **the first data structure** (page 4; page 5; page 21; page 22;

page 25; page 26; page 30) **comprising database tables** (page 4; page 5; page 21; page 22; page 25;

page 26; page 30), **procedure results from logic calls** (page 4; page 5; page 21; page 22; page 25;

page 26; page 30) **and status/error messages** (page 4; page 5; page 21; page 22; page 25; page 26; page

30); **translating application requests** (page 4; page 5; page 21; page 22; page 25; page 26; page 30)

from the application into a delimited second data structure (page 4; page 5; page 21; page

22; page 25; page 26; page 30), **stored in a memory** (page 4; page 5; page 21; page 22; page 25; page 26;

page 30), **the second data structure** (page 4; page 5; page 21; page 22; page 25; page 26; page 30)

comprising an element for each of the application requests (page 4; page 5; page 21; page

22; page 25; page 26; page 30); **generating a stream of text based** (page 4; page 5; page 21; page 22;

page 25; page 26; page 30), **data in an Extensible Markup Language (XML) format** (page 4;

page 5; page 21; page 22; page 25; page 26; page 30) **from the second data structure** (page 4; page 5;

page 21; page 22; page 25; page 26; page 30).

5.   Applicant argues (under REMARKS section) that, BEA WebLogic does not teach

"A method comprising: in a server, receiving a stream of text-based data in an

Extensible Markup Language (XML) format; parsing the stream into request statements;

and intercepting the request statements prior to execution and applying additional logic

based on a type or content of the request."

        Examiner respectfully disagrees. BEA WebLogic teaches **a method comprising:**

in a server, receiving a stream of text-based data in an Extensible Markup

Language (XML) format (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly*

write XML documents to the appropriate output stream. In the code segment below StockClient generates XML

based on user input that defines a stock trade transaction." ... // send xml to servlet

    pw.println ("<?xml version='1.0' ?>") ;

    // set document type declaration

    pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL

      + "stocktrade.dtd'>");

    // set stock trade instructions

    pw.print("<stocktrade ");

    pw.print("action = "+ (buy ? "'buy' : : "'sell' "));

    pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));

    pw.print("numshares = '"+ line +"'");

    pw.println(" />");"

...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document has been generated, you simply write the document to the appropriate output stream. " The preceding text excerpts clearly indicate that a user most likely through a web browser application/ console application sends requests for some stock transaction data to StockClient application. StockClient collects all of the necessary request information e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to StockServlet application through an output stream. The XML document includes elements i.e., data enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a structured XML request document in the first system to be sent to a second system through a stream, conversion of application requests.) (Page 4; page 21); **parsing the stream into request statements** (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream. " ... "In this example, StockServlet acts as a mediator between StockClient and TraderBean. StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to execute the trade. StockServlet then receives the result of the trade from TraderBean, generates XML in the format of traderresult.dtd, and sends the XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating SAX parser and displays the results to the console. " ... "Session Bean *Pure business "logic" bean *Contain no data *Similar to a stored procedure, but able to take advantage of the object services. Entity Bean *Represent a row in the table: "data" bean *A business object with a unique business identity *Identify determined by the primary key. Combine entity and session Beans as appropriate. An example using an order entry system: *Customer entity bean: the customer record *Order entity bean:the service order *Order process session bean: knows the process for completing an order: 1. Check customer credit limit 2. Check availability of stock 3. Place order, record (commit) a transaction. " ... "Though ejbStore() is called before the EJBean is actually written to the database, in the event an exception is raised while writing to the database, the framework will assume that the EJBean has been modified, regardless of the setting of the flag. "* The preceding text excerpts clearly indicate that the second system i.e., the system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the

StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the data i.e., processes or applies some business logic to the input data of the transaction. The result is then sent back to the first system, i.e., the system with StockClient. StockClient parses the result from the transaction result and shows to the user. Therefore whether it is a stock transaction or it is just getting a stock quote from the server, the TraderBean (whether it is a session or entity bean) has to interact with a persistent storage. Whether the TraderBean is a session or entity bean it performs some **business logic** for the transaction. The transaction result may be just getting the stock quote from the persistent storage i.e., a database table. The transaction result may comprise **failure/error/exception** to commit the transaction e.g., the requested stock is not available. The result may comprise an exception raised during the usage of the business logic of the EJBean. The EJBean i.e., TraderBean sends the **application result i.e., first data structure** to the StockServlet. The StockServlet converts the result into a **delimited XML structure i.e., second data structure** to the StockClient, which stores application results for the StockClient requested data elements.) (page 4; page 5; page 21; page 22; page 25; page 26; page 30); **and intercepting the request statements prior to execution** (page 4; page 5; page 21; page 22; page 25; page 26; page 30) **and applying additional logic** (page 4; page 5; page 21; page 22; page 25; page 26; page 30) **based on a type or content of the request** (i.e., if it is a buy, the traderBean performs a buy transaction or if it is a sell the traderBean performs a sell transaction.) (page 4; page 5; page 21; page 22; page 25; page 26; page 30).

6.   Any other arguments by the applicant are more limiting than the claimed language.

7.   Applicant's amendment necessitated new grounds of rejection. Examiner asserts that BEA WebLogic teaches Applicant's invention.

Claim Rejections - 35 USC § 102

8.   The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that

form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (a) the invention was known or used by others in this country, or patented or described in a printed publication in this or a foreign country, before the invention thereof by the applicant for a patent.

9.   Claims 1,4-11,13-17,20-21,23-24,26-27 are rejected under 35 U.S.C. 102(a) as

being anticipated by BEA (WebLogic Server documentation, copyright 1997-2000 and

last updated 2/9/2000 and BEA hereinafter).

      As to claim 1, BEA WebLogic teaches a method comprising, in a network (i.e.,

"Extensible Markup Language (XML) is a markup language for documents containing structured information. It is a

simplified version of Standard Generalized Markup Language (SGML) and has become an industry standard for

delivering content on the Internet.") (page 1; page 2), encapsulating data requests (i.e., "In the XML over

HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream. In

the code segment below StockClient generates XML based on user input that defines a stock trade transaction." ...

```
// send xml to servlet
    pw.println ("<?xml version='1.0' ?>") ;
    // set document type declaration
    pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL
        +"stocktrade.dtd'>");
    // set stock trade instructions
    pw.print("<stocktrade ");
    pw.print("action = "+ (buy ? "'buy' : : "'sell' "));
    pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));
    pw.print("numshares = '"+ line +"'");
    pw.println(" />");"
```

...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document has been generated, you simply write the document to the appropriate output stream." The preceding text excerpts clearly indicate that a user most likely through a web browser application/ console application sends requests for some stock transaction data to StockClient application. StockClient collects all of the necessary request information e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to StockServlet application through an output stream. The XML document includes elements i.e., data enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a structured XML request document in the first system to be sent to a second system through a stream, encapsulation.) (Page 4; page 21) generated by an application in a first system (i.e. the system with the StockClient application) (page 4), encapsulating (page 4) comprising generating an Extensible Markup Language (XML) structure (page 4) for each data request (page 4) and converting the XML structure (page 4) to an XML request (page 4), the XML structure (page 4) comprising a variable stream (page 4) of data stored in memory (i.e., "The DOM API provides methods for building and modifying entire XML documents in memory) (page 4) of the first system (page 4), the stream (page 4) including an XML element (page 4) for each request (page 4); transferring (i.e., StockClient connects to the StockServlet and sends an XML document via the POST method. The following code segment from StockClient obtains the URL to the servlet from the argument list, opens a URL connection, obtains the output stream from the connection, and prints the first couple of lines of the XML document to the output stream." The preceding text excerpts clearly indicate that StockClient transfers the request document to the StockServlet.) (page 6) the encapsulated (page 4) data requests (page 4) to a second system (page 4); executing (i.e., "In this example, StockServlet acts as a mediator between StockClient and TraderBean. StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to execute the trade. StockServlet

then receives the result of the trade from TraderBean, generates XML in the format of traderresult.dtd, and sends the

XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating SAX parser

and displays the results to the console." The preceding text excerpts clearly indicate that the second system i.e., the

system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the

StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the data i.e., processes the

transaction. The result is then sent back to the first system, i.e., the system with StockClient. StockClient parses the

result from the transaction result and shows to the user. The stock transaction is a persistent transaction. Therefore,

the TraderBean after processing the transaction persists/ commits the data in a persistent storage e.g., a database.)

(page 21; page 22; page 26; page 30) the converted application requests (Page 4; page 21) in the

second system (Page 4; page 21) into request statements (page 21; page 22; page 26); and

executing (page 21; page 22; page 26) the request statements (page 21; page 22; page 26) in the

second system (page 21; page 22; page 26).

As to claim 4, BEA teaches the XML element (i.e. the data is delimited by a tag and called an

element) (page 2) is a class object (i.e. instantiated objects of Java classes in Java client or Java server

applications e.g. StockClient) (page 5; page 21) whose data is stored to generate XML (page 5; page

21).

As to claim 5, BEA teaches that the XML element (i.e. the data is delimited by a tag and

called an element) (page 2) includes data from a data set object (i.e. instantiated objects of Java classes

in Java client or Java server applications e.g. StockClient) (page 5; page 21).

As to claim 6, BEA teaches that the data set object (i.e. instantiated objects of Java classes

in Java client or Java server applications e.g. StockServlet) (page 5; page 21) includes table dictionaries,

column names and data from record sets, and stored procedure parameters(i.e., *"In the*

XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output

stream." ... "In this example, StockServlet acts as a mediator between StockClient and TraderBean. StockClient

accepts user input to define a stock trade transaction. This data is used to generate XML in the format defined by

stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to RequestHandler

to parse the XML using a validating SAX parser and operates on TraderBean to execute the trade. StockServlet then

receives the result of the trade from TraderBean, generates XML in the format of traderresult.dtd, and sends the

XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating SAX parser

and displays the results to the console." ... "Session Bean *Pure business "logic" bean *Contain no data *Similar

to a stored procedure, but able to take advantage of the object services. Entity Bean *Represent a row in the table:

"data" bean *A business object with a unique business identity *Identify determined by the primary key. Combine

entity and session Beans as appropriate. An example using an order entry system: *Customer entity bean: the

customer record *Order entity bean:the service order *Order process session bean: knows the process for

completing an order: 1. Check customer credit limit 2. Check availability of stock 3. Place order, record (commit) a

transaction." ... "Though ejbStore() is called before the EJBean is actually written to the database, in the event an

exception is raised while writing to the database, the framework will assume that the EJBean has been modified,

regardless of the setting of the flag." The preceding text excerpts clearly indicate that the second system i.e., the

system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the

StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the data i.e., processes the

transaction. The result is then sent back to the first system, i.e., the system with StockClient. StockClient parses the

result from the transaction result and shows to the user. Therefore whether it is a stock transaction or it is just getting

a stock quote from the server, the TraderBean (whether it is a session or entity bean) has to interact with a persistent

storage. Whether the TraderBean is a session or entity bean it performs some business logic for the transaction. The

transaction result may be just getting the stock quote from the persistent storage i.e., a database table. The

transaction result may comprise **failure/error/exception** to commit the transaction e.g., the requested stock is not

available. The result may comprise an exception raised during the usage of the business logic of the EJBean. The

EJBean i.e., TraderBean sends the **application result i.e., first data structure** to the StockServlet. The StockServlet

converts the result into a **delimited XML structure i.e., second data structure** to the StockClient, which stores

application results for the StockClient requested data elements.) (page 4; page 5; page 21; page 22; page 25; page

26; page 30).

As to claim 7, BEA teaches that transferring includes a text transmission protocol

(i.e. HTTP) (page 4).

As to claim 8, BEA teaches that the text transmission protocol is Hypertext

Transfer Protocol (HTTP) (page 4).

As to claim 9, BEA teaches de-encapsulating the encapsulated data requests by

parsing into request statements (page 4; page 5; page 21; page 22; page 25; page 26; page 30); and

executing the request statements (i.e. StockServlet operates/executes on TraderBean to execute the

trade) (page 4; page 5; page 21; page 22; page 25; page 26; page 30).

As to claim 10, BEA teaches translating responses from the executed request

statements into an XML format (page 4; page 5; page 21; page 22; page 25; page 26; page 30); and

sending the XML formatted responses to the first system (i.e. the StockServlet sends the requested

result data back to StockClient in XML format for StockClient to process the data or whatever it needs to do with

that data) (page 5; page 21).

As to claim 11, BEA WebLogic teaches a distributed application (i.e., "Extensible

Markup Language (XML) is a markup language for documents containing structured information. It is a simplified

version of Standard Generalized Markup Language (SGML) and has become an industry standard for delivering

content on the Internet.") (page 1; page 2) method comprising: converting application requests (i.e.,

"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate

output stream. In the code segment below StockClient generates XML based on user input that defines a stock trade

transaction." ... // send xml to servlet

pw.println ("<?xml version='1.0' ?>") ;

// set document type declaration

pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL

+ "stocktrade.dtd'>");

// set stock trade instructions

pw.print("<stocktrade ");

pw.print("action = "+ (buy ? "'buy' : : "'sell' "));

pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));

pw.print("numshares = '"+ line +"'");

pw.println(" />");"

...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document

has been generated, you simply write the document to the appropriate output stream." The preceding text excerpts

clearly indicate that a user most likely through a web browser application/ console application sends requests for

some stock transaction data to StockClient application. StockClient collects all of the necessary request information

e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to

StockServlet application through an output stream. The XML document includes elements i.e., data

enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a

structured XML request document in the first system to be sent to a second system through a stream, conversion of

application requests.) (Page 4; page 21) in a first system (Page 4; page 21), converting comprising

generating a data structure (Page 4; page 21) for storing data and parameters related to an

application that produced the application requests (i.e., the XML structure created by StockClient

stores all of the transaction request data sent by the user through a console application/ a web browser) (Page 4; page

21), translating the application requests (Page 4; page 21) into a standardized delimited data

structure (Page 4; page 21) in conjunction with the data structure (Page 4; page 21) into a stream

(i.e., the output stream) (Page 4; page 21) of text based data utilizing a Extensible Markup

Language (XML) format (Page 4; page 21); transmitting (i.e., *"StockClient connects to the StockServlet*

and sends an XML document via the POST method. The following code segment from StockClient obtains the URL

to the servlet from the argument list, opens a URL connection, obtains the output stream from the connection, and

prints the first couple of lines of the XML document to the output stream." The preceding text excerpts clearly

indicate that StockClient transmits/ transfers the request document to the StockServlet.) (page 6) the converted

application requests (Page 4; page 21) to a second system (Page 4; page 21) over a network

(page 1; page 2); parsing (i.e., *"In this example, StockServlet acts as a mediator between StockClient and*

TraderBean. StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in

the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an

instance to RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to execute

the trade. StockServlet then receives the result of the trade from TraderBean, generates XML in the format of

traderresult.dtd, and sends the XML back to StockClient via a HTTPServletResponse. StockClient parses the XML

using a validating SAX parser and displays the results to the console." The preceding text excerpts clearly indicate

that the second system i.e., the system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction

request data from the StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the

data i.e., processes the transaction. The result is then sent back to the first system, i.e., the system with StockClient.

StockClient parses the result from the transaction result and shows to the user. The stock transaction is a persistent

transaction. Therefore, the TraderBean after processing the transaction persists/ commits the data in a persistent

storage e.g., a database.) (page 21; page 22; page 26) the converted application requests (Page 4; page

21) in the second system (Page 4; page 21) into request statements (page 21; page 22; page 26);

and executing (page 21; page 22; page 26) the request statements (page 21; page 22; page 26) in the

second system (page 21; page 22; page 26).

As to claim 13, BEA teaches that the parsing comprises: breaking down the

converted application requests (page 4; page 5; page 21; page 22; page 25; page 26; page 30) to an

executable command format (page 4; page 5; page 21; page 22; page 25; page 26; page 30) utilizing

data and parameters related to an application (i.e. StockServlet operates/executes on TraderBean to

execute the trade) (page 5; page 21).

As to claim 14, BEA teaches that executing further comprises evaluating

executable commands prior to execution in the second system (i.e. StockServlet

operates/executes on TraderBean to execute the trade; The StockServlet or the TraderBean has to look at/evaluate

the sent parameters to know what it is supposed to do.) (page 5; page 21).

As to claim 15, BEA teaches that executing further comprises evaluating results

generated by the executable commands (i.e. StockServlet operates/executes on TraderBean to execute

the trade; The StockServlet or the TraderBean has to look at the sent parameters to know what it is supposed to do

with those parameters. After it executes whatever it needs to do, it has to look at/ evaluate the data that it is about to

send back to the client) (page 5; page 21).

As to claim 16, BEA teaches converting the results into a stream of text based

data in a standardized XML format (i.e. the input stream or output stream through which an application

receives the XML data; an application transferring data writes to an output stream and an application receiving data

reads data from an input stream.) (page 4); and transmitting the converted results over the network

to the first system (page 5; page 21).

As to claims 17 and 26, BEA WebLogic teaches an application server (i.e., the

WebLogic server with TraderBean) (page 21; page 22; page 26) method comprising: generating (i.e., *"In*

the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate

output stream." ... *"In this example, StockServlet acts as a mediator between StockClient and TraderBean.*

StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in the format

defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to

RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to execute the trade.

StockServlet then receives the result of the trade from TraderBean, generates XML in the format of traderresult.dtd,

and sends the XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating

SAX parser and displays the results to the console." ... *"Session Bean *Pure business "logic" bean *Contain no*

data *Similar to a stored procedure, but able to take advantage of the object services. Entity Bean *Represent a row

in the table: "data" bean *A business object with a unique business identity *Identify determined by the primary

key. Combine entity and session Beans as appropriate. An example using an order entry system: *Customer entity

bean: the customer record *Order entity bean:the service order *Order process session bean: knows the process for

completing an order: 1. Check customer credit limit 2. Check availability of stock 3. Place order, record (commit) a

transaction." ... *"Though ejbStore() is called before the EJBean is actually written to the database, in the event an*

exception is raised while writing to the database, the framework will assume that the EJBean has been modified,

regardless of the setting of the flag." The preceding text excerpts clearly indicate that the second system i.e., the

system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the

StockClient. The second system parses the data and then the EJB TraderBean executes the data i.e., processes the

transaction. The result is then sent back to the first system, i.e., the system with StockClient. StockClient parses the

result from the transaction result and shows to the user. Therefore whether it is a stock transaction or it is just getting

a stock quote from the server, the TraderBean (whether it is a session or entity bean) has to interact with a persistent

storage. Whether the TraderBean is a session or entity bean it performs some business logic for the transaction. The

transaction result may be just getting the stock quote from the persistent storage i.e., a database table. The

transaction result may comprise **failure/error/exception** to commit the transaction e.g., the requested stock is not

available. The result may comprise an exception raised during the usage of the business logic of the EJBean. The

EJBean i.e., TraderBean sends the **application result i.e., first data structure** to the StockServlet. The StockServlet

converts the result into **a delimited XML structure i.e., second data structure** to the StockClient, which stores

application results for the StockClient requested data elements.) (page 4; page 5; page 21; page 22; page 25; page

26; page 30) a first data structure (page 4; page 5; page 21; page 22; page 25; page 26; page 30) for storing

data and parameters (page 4; page 5; page 21; page 22; page 25; page 26; page 30) related to an

application residing in the server (page 4; page 5; page 21; page 22; page 25; page 26; page 30), the

first data structure (page 4; page 5; page 21; page 22; page 25; page 26; page 30) comprising database

tables (page 4; page 5; page 21; page 22; page 25; page 26; page 30), procedure results from logic calls

(page 4; page 5; page 21; page 22; page 25; page 26; page 30) and status/error messages (page 4; page 5;

page 21; page 22; page 25; page 26; page 30); translating application requests (page 4; page 5; page 21;

page 22; page 25; page 26; page 30) from the application into a delimited second data structure

(page 4; page 5; page 21; page 22; page 25; page 26; page 30), stored in a memory (page 4; page 5; page 21;

page 22; page 25; page 26; page 30), the second data structure (page 4; page 5; page 21; page 22; page 25;

page 26; page 30) comprising an element for each of the application requests (page 4; page 5;

page 21; page 22; page 25; page 26; page 30); generating a stream of text based (page 4; page 5; page

21; page 22; page 25; page 26; page 30), data in an Extensible Markup Language (XML) format

(page 4; page 5; page 21; page 22; page 25; page 26; page 30) from the second data structure (page 4;

page 5; page 21; page 22; page 25; page 26; page 30).

As to claim 18, BEA teaches that the first data structure (page 4; page 5; page 21; page 22; page 25; page 26; page 30) includes database tables (page 4; page 5; page 21; page 22; page 25; page 26; page 30), procedure results from logic calls and status/error messages (i.e. the TraderBean trading results are sent back to the StockClient via the StockServlet) (page 4; page 5; page 21; page 22; page 25; page 26; page 30).

As to claim 20, BEA teaches that the element (i.e. the data is delimited by a tag and called an element) (page 2) is a class object (i.e. instantiated objects of Java classes in Java client or Java server applications e.g. StockClient) (page 5; page 15; page 21).

As to claims 21 and 27, BEA WebLogic teaches a method comprising: in a server, receiving a stream of text-based data in an Extensible Markup Language (XML) format (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream. In the code segment below StockClient generates XML based on user input that defines a stock trade transaction."* ... // send xml to servlet

```
pw.println ("<?xml version='1.0' ?>") ;
// set document type declaration
pw.println("<!DOCTYPE stocktrade SYSTEM '"+serverURL
    +"stocktrade.dtd'>");
// set stock trade instructions
pw.print("<stocktrade ");
pw.print("action = "+ (buy ? "'buy' : : "'sell' "));
pw.print("symbol = "+ (webl ? "'WEBL' " : "'INTL' "));
pw.print("numshares = '"+ line +"'");
```

pw.println(" />");"

...

"The DOM API provides methods for building and modifying entire XML documents in memory. Once a document has been generated, you simply write the document to the appropriate output stream. " The preceding text excerpts clearly indicate that a user most likely through a web browser application/ console application sends requests for some stock transaction data to StockClient application. StockClient collects all of the necessary request information e.g., buy, sell, stock symbol name, number of shares etc. and generates a structured XML request to send it to StockServlet application through an output stream. The XML document includes elements i.e., data enclosed/delimited by '<' and "/>", containing the transaction data. Applicant calls this process of creating a structured XML request document in the first system to be sent to a second system through a stream, conversion of application requests.) (Page 4; page 21); parsing the stream into request statements (i.e., *"In the XML over HTTP example, StockClient and StockServlet explicitly write XML documents to the appropriate output stream. "* ... *"In this example, StockServlet acts as a mediator between StockClient and TraderBean. StockClient accepts user input to define a stock trade transaction. This data is used to generate XML in the format defined by stocktrade.dtd and the XML is sent to StockServlet via a POST. StockServlet creates an instance to RequestHandler to parse the XML using a validating SAX parser and operates on TraderBean to execute the trade. StockServlet then receives the result of the trade from TraderBean, generates XML in the format of traderresult.dtd, and sends the XML back to StockClient via a HTTPServletResponse. StockClient parses the XML using a validating SAX parser and displays the results to the console. "* ... *"Session Bean *Pure business "logic" bean *Contain no data *Similar to a stored procedure, but able to take advantage of the object services. Entity Bean *Represent a row in the table: "data" bean *A business object with a unique business identity *Identify determined by the primary key. Combine entity and session Beans as appropriate. An example using an order entry system: *Customer entity bean: the customer record *Order entity bean:the service order *Order process session bean: knows the process for completing an order: 1. Check customer credit limit 2. Check availability of stock 3. Place order, record (commit) a transaction. "* ... *"Though ejbStore() is called before the EJBean is actually written to the database, in the event an exception is raised while writing to the database, the framework will assume that the EJBean has been modified, regardless of the setting of the flag. "* The preceding text excerpts clearly indicate that the second system i.e., the

system/ WebLogic Server with StockServlet, TraderBean etc. receives the transaction request data from the StockClient. The second system **parses** the data and then the EJB TraderBean **executes** the data i.e., processes or applies some business logic to the input data of the transaction. The result is then sent back to the first system, i.e., the system with StockClient. StockClient parses the result from the transaction result and shows to the user. Therefore whether it is a stock transaction or it is just getting a stock quote from the server, the TraderBean (whether it is a session or entity bean) has to interact with a persistent storage. Whether the TraderBean is a session or entity bean it performs some **business logic** for the transaction. The transaction result may be just getting the stock quote from the persistent storage i.e., a database table. The transaction result may comprise **failure/error/exception** to commit the transaction e.g., the requested stock is not available. The result may comprise an exception raised during the usage of the business logic of the EJBean. The EJBean i.e., TraderBean sends the **application result i.e., first data structure** to the StockServlet. The StockServlet converts the result into **a delimited XML structure i.e., second data structure** to the StockClient, which stores application results for the StockClient requested data elements.) (page 4; page 5; page 21; page 22; page 25; page 26; page 30); and intercepting the request statements prior to execution (page 4; page 5; page 21; page 22; page 25; page 26; page 30) and applying additional logic (page 4; page 5; page 21; page 22; page 25; page 26; page 30) based on a type or content of the request (i.e., if it is a buy, the traderBean performs a buy transaction or if it is a sell the traderBean performs a sell transaction.) (page 4; page 5; page 21; page 22; page 25; page 26; page 30).

As to claim 23, BEA teaches that executing further comprises applying additional logic to responses generated from executing the request statements (page 4; page 5; page 7; page 8; page 21).

As to claim 24, BEA teaches converting responses generated from each of the executed request statements into an XML format (page 4; page 5; page 7; page 8; page 21).

Conclusion

10.   **THIS ACTION IS MADE FINAL.** See MPEP § 706.07(a). Applicant is reminded of the extension of time policy as set forth in 37 CFR 1.136(a).

A shortened statutory period for reply to this final action is set to expire THREE MONTHS from the mailing date of this action. In the event a first reply is filed within TWO MONTHS of the mailing date of this final action and the advisory action is not mailed until after the end of the THREE-MONTH shortened statutory period, then the shortened statutory period will expire on the date the advisory action is mailed, and any extension fee pursuant to 37 CFR 1.136(a) will be calculated from the mailing date of the advisory action. In no event, however, will the statutory period for reply expire later than SIX MONTHS from the date of this final action.
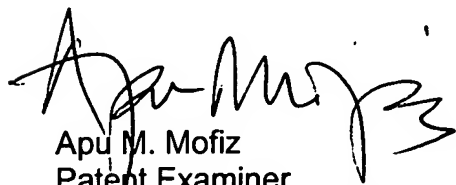
Points of Contact

11.   Any inquiry concerning this communication or earlier communications from the examiner should be directed to Apu M. Mofiz whose telephone number is (571) 272-4080. The examiner can normally be reached on Monday – Thursday 8:00 A.M. to 4:30 P.M.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Dov Popovici can be reached at (571) 272-4083. The fax numbers for the group is (703) 872-9306.

Any inquiry of a general nature or relating to the status of this application should

be directed to the Group receptionist whose telephone number is (703) 305-9600.

Apu M. Mofiz
Patent Examiner
Technology Center 2100

March 29,2005